2

VMs for Portability: BCPL

2.1 Introduction

BCPL is a high-level language for systems programming that is intended to be as portable as possible. It is now a relatively old language but it contains most syntactic constructs found in contemporary languages. Indeed, C was designed as a BCPL derivative (C can be considered as a mixture of BCPL and Algol68 plus some *sui generis* features). BCPL is not conventionally typed. It has one basic data type, the machine word. It is possible to extract bytes from words but this is a derived operation. All entities in BCPL are considered either to be machine words or to require a machine word or a number of machine words. BCPL supports addresses and assumes that they can fit into a single word. Similarly, it supports vectors (one-dimensional arrays) which are sequences of words (multi-dimensional arrays must be explicitly programmed in terms of vectors of pointers to vectors). Routines (procedures and functions) can be defined in BCPL and are represented as pointers to their entry points. Equally, labels are addresses of sequences of instructions.

BCPL stands for "Basic CPL", a subset of the CPL language. CPL was an ambitious lexically scoped, imperative procedural programming language designed by Strachey and others in the mid-1960s as a joint effort involving Cambridge and London Universities. CPL contained all of the most advanced language constructs of the day, including polymorphism. There is a story that the compiler was too large to run on even the biggest machines available in the University of London! Even though it strictly prefigures the structured programming movement, BCPL contains structured control constructs (commands) including two-branch conditionals, switch commands, structured loops with structured exits. It also supports statement formulæ similar to those in FORTRAN and the original BASIC. Recursive routines can be defined. BCPL does support a goto command. Separate compilation is supported in part by the provision of a "global vector", a vector of words that contains pointers to externally defined routines. BCPL is lexically scoped. It implements call-by-value semantics for routine parameters. It also permits higher-order

programming by permitting routine names to be assigned to variables (and, hence, passed into and out of routines).

BCPL was intended to be portable. Portability is achieved by bootstrapping the runtime system a number of times so that it eventually implements the compiler's output language. This language is called *OCODE*. OCODE is similar to a high-level assembly language but is tailored exactly to the intermediate representation of BCPL constructs. OCODE was also defined in such a way that it could be translated into the machine language of most processors. Associated with OCODE is an OCODE machine that, once implemented, executes OCODE, hence compiled BCPL. The implementation of an abstract machine for OCODE is relatively straigthforward.

In the book on BCPL [45], Richards and Whitby-Strevens define a second low-level intermediate language called *Intcode*. Intcode is an extremely simple language that can be used to bootstrap OCODE. More recently, Richards has defined a new low-level bootstrap code called *Cintcode*. The idea is that a fundamental system is first written for Intcode/Cintcode. This is then used to bootstrap the OCODE evaluator. The definition of the Intcode and Cintcode machines is given in the BCPL documentation. The BCPL system was distributed in OCODE form (more recent versions distribute executables for standard architectures like the PC under Linux). At the time the book was published, an Intcode version of the system was required to bootstrap a new implementation.

The virtual machines described below are intended, therefore, as an aid to portability. The definitions of the machines used to implement OCODE and Intcode/Cintcode instructions include definitions of the storage structures and layout required by the virtual machine, as well as the instruction formats and state transitions.

The organisation of this chapter is as follows. We will focus first on BCPL and its intermediate languages OCODE and Intcode/Cintcode (Cintcode is part of the current BCPL release and access to the documentation is relatively easy). We will begin with a description of the OCODE machine. This description will start with a description of the machine's organisation and then we move on to a description of the instruction set. The relationship between OCODE instructions and BCPL's semantics will also be considered. Then, we will examine Cintcode and its abstract machine. Finally, we explain how BCPL can be ported to a completely new architecture.

2.2 BCPL the Language

In this section, the BCPL language is briefly described.

BCPL is what we would now see as a relatively straightforward procedural language. As such, it is based around the concept of the procedure. BCPL provides three types of procedural abstraction:

- Routines that update the state and return no value;

- Routines that can update the state and return a single value;
- Routines that just compute a value.

The first category refers to procedures proper, while the second corresponds to the usual concept of function in procedural languages. The third category corresponds to the single-line functions in FORTRAN and in many BASIC dialects. Each category permits the programmer to pass parameters, which are called by value.

BCPL also supports a variety of function that is akin to the so-called "formula function" of FORTRAN and BASIC. This can be considered a variety of macro or open procedure because it declares no local variables.

BCPL supports a variety of state-modifying constructs. As an imperative language, it should be obvious that it contains an assignment statement. Assignment in BCPL can be simple or multiple, so the following are both legal:

```
x := 0;
x, y := 1, 2;
```

It is worth noting that terminating semicolons are optional. They are mandatory if more than one command is to appear on the same line as in:

```
x := 0; y := 2
```

Newline, in BCPL, can also be used to terminate a statement. This is a nice feature, one found in only a few other languages (Eiffel and Imp, a language used in the 1970s at Edinburgh University).

Aside from this syntactic feature, the multiple assignment gives a clue that the underlying semantics of BCPL are based on a stack.

In addition, it contains a number of branching constructs:

- IF ... DO.[1] This is a simple test. If the test is true, the code following the DO is executed. If the test is false, the entire statement is a no-operation.
- UNLESS ... DO. This is syntactic sugar for IF NOT ... DO. That is, the code following the DO is executed if the test fails.
- TEST ... THEN ... ELSE. This corresponds to the usual if then else in most programming languages.
- SWITCHON. This is directly analogous to the case statement in Pascal and its descendants and to the switch statement in C and its derivatives. Cases are marked using the CASE keyword. Cases run into each other unless explicitly broken. There is also a an optional default case denoted by a keyword. Each case is implicitly a block.

In general, the syntax word do can be interchanged with then. In the above list, we have followed the conventions of BCPL style.

BCPL contains a number of iterative statements. The iterative statements are accompanied by structured ways to exit loops.

[1] Keywords must be in uppercase, so the convention is followed here.

BCPL has a goto, as befits its age.

BCPL statements can be made to return values. This is done using the pair of commands VALOF and RESULTIS. The VALOF command introduces a block from which a value is returned using the RESULTIS command; there can be more than one RESULTIS command in a VALOF block. The combination of VALOF and RESULTIS is used to return values from functions. The following is a BCPL procedure:

```
LET Add.Global (x) BE
$(
  globl := globl + x;
$)
```

The following is a BCPL functional routine:

```
LET Global.Added.Val (x) =
$(
  VALOF $(
    RESULTIS(x+globl);
  $)
$)
```

From this small example, it can be seen that the body of a procedure is marked by the BE keyword, while functional routines are signalled by the equals sign and the use of VALOF and RESULTIS (BCPL is case-sensitive).

BCPL is not conventionally typed. It has only one data type, the machine word, whose size can change from machine to machine. The language also contains operators that access the bytes within a machine word. Storage is allocated by the BCPL compiler in units of one machine word. The language contains an operator that returns the address of a word and an operator that, given an address, returns the contents of the word at that address (dereferencing).

BCPL supports structured types to a limited extent. It permits the definition of vectors (single-dimension arrays of words). It also has a table type. Tables are vectors of words that are indexed by symbolic constants, not by numerical values. In addition, it is possible to take the address of a routine (procedure or function); such addresses are the *entry points* of the routines (as in C). The passing of routine addresses is the method by which BCPL supports higher-order routines (much as C does).

It also permits the definition of symbolic constants. Each constant is one machine word in length.

BCPL introduces entities using the LET syntax derived from ISWIM. For example, the following introduces a new variable that is initialised to zero:

```
LET x := 0 IN
```

The following introduces a constant:

```
LET x = 0 IN
```

Multiple definitions are separated by the AND keyword (logical conjunction is represented by the "&" symbol) as in:

```
LET x := 0
AND y = 0
IN
```

Routines are also introduced by the LET construct.

Variables and constants can be introduced at the head of any block.

In order to support separate compilation and to ease the handling of the runtime library, a *global vector* is supported. This is a globally accessible vector of words, in which the first few dozen entries are initialised by the runtime system (they are initialised to library routine entry points and to globally useful values). The programmer can also assign to the global vector at higher locations (care must be taken not to assign to locations used by the system). These are the primary semantic constructs of BCPL. Given this summary, we can now make some observations about the support required by the virtual machine (the OCODE machine).

2.3 VM Operations

The summary of BCPL above was intended to expose the major constructs. The identification of major constructs is important for the design of a virtual machine which must respect the semantics of the language as well as providing the storage structures required to support the language.

At this stage, it should be clear that a BCPL machine should provide support for the primitive operations needed for the manipulation of data of all primitive types. The virtual machine support for them will be in the form of instructions that the machine will directly implement. In BCPL, this implies that the virtual machine must support operations on the word type: arithmetic operations, comparisons and addressing. Byte-based operations can either be provided by runtime library operations or by instructions in the virtual machine; BCPL employs the latter for the reason that it is faster and reduces the size of the library. In addition, BCPL supports vectors on the stack; they must also be addressed when designing an appropriate virtual machine.

The values manipulated by these operations must be stored somewhere: a storage area, particularly for temporary and non-global values must be provided. Operations are required for manipulating this storage area. Operations are also required to load values from other locations and to store them as results. More than one load operation might be required (in a more richly typed language, this might be a necessity) and more than one store operation might be required. It is necessary to look at the cases to determine what is required.

BCPL employs static scoping. The compiler can be relied upon to verify that variables, etc., are not required. Static scoping requires a stack-like mechanism for the storage of variables. The virtual machine is, therefore, built around a stack. Operations are required to allocate and free regions of stack at routine entry and exit; the return of results can also be implemented by means of stack allocation and addressing. The compiler generates instructions that allocate and free the right amount of stack space; it also generates instructions to handle returned values and the adjustment of the stack when routines return. Evaluation of expressions can be performed on the stack, so we now are in a position to define the instructions for data manipulation.

With expressions out of the way, the following families of construct must be handled by the compiler and OCODE instructions generated to implement them:

- Control constructs, in particular, conditionals, iteration, jumps;
- Assignment;
- Routine call and return;
- Parameter passing and value return from routines and valof.

Note that we assume that sequencing is handled implicitly by the compiler.

Control structure is handled, basically, by means of labels and jumps. There are clear translations between most of the control structures and label-jump combinations. The problem cases are FOR and SWITCHON. The former is problematic because it requires counters to be maintained and updated in the right order; the latter because the best implementation requires a jump table.

Assignment is a relatively straightforward matter (essentially, push a value onto the stack and pop it off to some address or other). Multiple assignment is also easy with a stack machine. The values are pushed onto the stack in some order (say left to right) and popped in the reverse order. Thus, the command:

```
p,q := 1, 2
```

has the intention of assigning 1 to p and 2 to q. This can be done by pushing 1, then 2 onto the stack and assigning them in reverse order. An interesting example of multiple assignment is:

```
p,q := q, p
```

Swap! It can be handled in exactly the manner just described.

Finally, we have routine calls and VALOF. There are many ways to implement routine calls. For software virtual machines, relatively high-level instructions can be used (although low-level instructions can also be employed). The OCODE machine provides special instructions for handling routine entry and exit, as will be seen.

BCPL is a call-by-value language, so the runtime stack can be directly employed to hold parameter values that are to be passed into the routine.

The VALOF ... RESULTIS combination can be handled in a variety of ways. One is to perform a source-to-source transformation. Another is to use the stack at runtime by introducing a new scope level. Variables local to the VALOF can be allocated on the runtime stack with the stack then being used for local values until the RESULTIS is encountered. An implementation for RESULTIS would be to collapse the stack to the point where the VALOF was encountered and then push the value to be returned onto the stack.

2.4 The OCODE Machine

In this section, the organisation of the OCODE machine is presented. BCPL is a procedural programming language that supports recursion. It requires a globally accessible vector of words to support separate compilation. It also requires a pool of space to represent global variables. The language also permits the use of (one-dimensional) vectors and tables (essentially vectors of words whose elements are indexed by symbolic identifiers, much like tables in assembly language). As a consequence, the OCODE machine must reserve space for a stack to support lexical scope and for recursion. The OCODE machine also needs space to hold the global vector and also needs a space to hold program instructions.

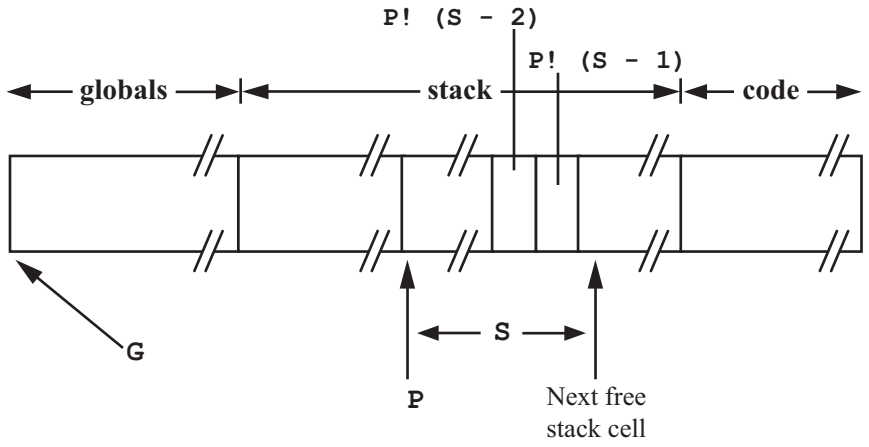

Fig. 2.1. *The OCODE machine organisation.*

The OCODE machine has three memory regions:

- The Global vector;
- The Stack (this is a *framed* stack);
- Storage for program code and static data.

The organisation of the OCODE machine is shown in Figure 2.1.

The *global vector* is used to store all variables declared global in the program. The global vector is a vector of words containing global variables; it also contains the entry points of routines declared in one module that are to be made visible in another. It is pointed to by the `G` register. The current stack frame is pointed to by the `P` register. The size of the current stack frame is always known at compilation time, so it need not be represented in code by a register.

There is also a special `A` register which is used to hold values returned by functions (see below).

Static variables, tables and string constants are stored in the program area. They are referenced by labels which are usually represented by the letter `L` followed by one or more digits.

The stack holds all dynamic (local) variables.

All variables are of the same size. That is, all variables are allocated the same amount of space in the store. For most modern machines they are 32- or 64-bits in length.

2.5 OCODE Instructions and their Implementation

In OCODE, instructions are represented as integers. Here, we will use only the mnemonic names in the interests of readability. It is important to note that the mnemonic form for instructions and labels must be converted into more fundamental representations when code is emitted by the compiler.

The size of the current stack frame is always known at compile time. When specifying instructions, a variable, `S`, is used to denote an offset from the start of the current stack frame. This is done only to show how much space is left in the current stack frame by the individual instructions.

When defining abstract machine instructions, an array notation will be employed. Thus, `P` is considered as a one-dimensional vector. `S` will still be a constant denoting the size of the current stack frame. Similarly, `G` will also be considered as an array.

The notation `P[S-1]` denotes the first free element on the stack.

2.5.1 Expression Instructions

The OCODE instructions that implement expressions do not alter the stack frame size. In the case of unary instructions, the operand is replaced on the top of the stack by the result of the instruction. In the case of binary operations, the stack element immediately beneath the top one is replaced by the result.

The instructions are mostly quite clear. Rather than enter into unnecessary detail, these instructions are summarised in Table 2.1 The table's middle column is a short English equivalent for the opcode.

Only the first instruction deserves any real comment. It is an instruction that considers the current top-of-stack element as a pointer into memory. It replaces the top-of-stack element by the object that it points to. This is the operation of dereferencing a pointer to yield an r-value.

Table 2.1. *OCODE expression instructions.*

Opcode	Description	Definition
RV	r-value	P[S-1] := cts([S-1])
ABS	absolute value	P[S-1] := abs(P[S-1])
NEG	unary minus	P[S-1] := -P[S-1]
NOT	logical negation	P[S-1] := ¬(P[S-1])
GETBYTE	extract byte	P[S-2] := P[S-2] gtb P[S-1]
MULT	multiply	P[S-2] := P[S-2] * P[S-1]
DIV	divide	P[S-2] := P[S-2] / P[S-1]
REM	remainder	P[S-2] := P[S-2] rem P[S-1]
PLUS	add	P[S-2] := P[S-2] + P[S-1]
MINUS	subtract	P[S-2] := P[S-2] - P[S-1]
EQ	equal	P[S-2] := P[S-2] = P[S-1]
NE	not equal	P[S-2] := P[S-2] ≠ P[S-1]
LS	less than	P[S-2] := P[S-2] < P[S-1]
GR	greater than	P[S-2] := P[S-2] > P[S-1]
LE	≤	P[S-2] := P[S-2] <= P[S-1]
GE	≥	P[S-2] := P[S-2] >= P[S-1]
LSHIFT	left shift	P[S-2] := P[S-2] << P[S-1]
RSHIFT	right shift	P[S-2] := P[S-2] >> P[S-1]
LOGAND	logical and	P[S-2] := P[S-2] and P[S-1]
LOGOR	logical or	P[S-2] := P[S-2] or P[S-1]
EQV	bitwise equal	P[S-2] := P[S-2] leq P[S-1]
NEQV	xor	P[S-2] := P[S-2] xor P[S-1]

Table 2.1 employs a notational convention that needs explanation:

- cts is the contents operation (dereferences its argument).
- abs is the absolute value of its argument.
- gtb is the getbyte operator.
- rem is integer remainder after division.
- and is logical and (conjunction).
- or is logical or (disjunction).
- leq is bitwise equivalence.
- xor is bitwise exclusive or (logical not-equivalence).
- $e_1 << e_2$ is left shift e1 by e2 bits.
- $e_1 >> e_2$ is right shift e1 by e2 bits.

Other than this, the "description" of each instruction is just an operation on the OCODE stack. In this and the following cases, the code equivalent is

included in the table; when defining virtual machines later in this book, this method will be used to indicate both "descriptions" and implementations of virtual machine instructions.

2.5.2 Load and Store Instructions

The load and store instructions, like those for expressions, should be fairly clear. The code equivalents are included in the right-hand column of Table 2.2. Each instruction is described (middle column of the table).

Table 2.2. *OCODE load and store instructions.*

Opcode	Description	Definition
LPn	load from P	P[S] := P[n]; S := S+1
LGn	load global	P[S] := G[n]; S := S+1
LL Ln	load label	P[S] := Ln; S := S+1
LL Pn	load address	P[S] := P[n]; S := S+1
LL Gn	load global addr	P[S] := G[n]; S := S+1
LLL Ln	load label addr	P[S] := Ln; S := S+1
SPn	store off P	P[n] := P[S]; S := S-1
SGn	store global	G[n] := P[S]; S := S-1
SL Ln	store at label	Ln:= P[S]; S := S-1
LF Ln	load function	P[S] := entry point Ln; S := S+1
LNn	load constant	P[S] := n; S := S+1
TRUE	true	P[S] := *true*; S := S+1
FALSE	false	P[S] := *false*; S := S+1
LSTR $n\, C_1 \ldots C_n$	load string	P[S] := "$C_1 \ldots C_n$"; S := S+1
STIND	store index	cts(P[S-1]) := P[S-2]; S := S-2
PUTBYTE	put byte	setbyte(P[S-2],P[S-1]) := P[S-3]; S := S-3

There is an instruction not included in Table 2.2 that appears in the OCODE machine specification in [44]. It is the QUERY instruction. It is defined as:

P[S] := *?*; S := S+1

Unfortunately, [44] does not contain a description of it. The remaining instructions have an interpretation that is fairly clear and is included in the table. It is hoped that the relatively brief description is adequate.

2.5.3 Instructions Relating to Routines

This class of instruction deals with routine entry (call) and return. When it compiles a routine, the OCODE compiler generates code of the following form:

```
ENTRY Li n C1 ... Cn
SAVE s
<body of routine>
ENDPROC
```

Here, `Li` is the label of the routine's entry point. For debugging purposes, the length of the routine's identifier is recorded in the code (this is `n` in the code fragment); the characters comprising the name are the elements denoted `C1` to `Cn`. The instructions in this category are shown in Table 2.3.

The `SAVE` instruction specifies the initial setting of the `S` register. The value of this is the save space size (3) plus the number of formal parameters. The save space is used to hold the previous value of `P`, the return address and the routine entry address. The first argument to a routine is always at the location denoted by 3 relative to the pointer `P` (some versions of BCPL have a different save space size, so the standard account is followed above).

The end of each routine is denoted by `ENDPROC`. This is a no-op which allows the code generator to keep track of nested procedure definitions.

The BCPL standard requires that arguments are allocated in consecutive locations on the stack. There is no *a priori* limit to the number of arguments that can be supplied. A typical call of the form:

```
E(E1, ..., En)
```

is compiled as follows (see Table 2.3). First, `S` is incremented to allocate space for the save space in the new stack frame. The arguments `E1` to `En` are compiled and then the code for `E`. Finally, either `FNAP` k or `RTAP` k instruction is generated, the actual one depending upon whether a function or routine call is being compiled. The value k is the distance between the old and new stack frames (i.e., the number of words or bytes between the start of the newly compiled stack frame and the start of the previous one on the stack).

Table 2.3. *OCODE instructions for routines.*

Opcode	Meaning
`ENTRY`	enter routine
`SAVE`	save locals
`ENDPROC`	end routine
`FNAP`k	apply function
`RNAP`k	apply procedure
`RTRN`	return from procedure
`FNRN`	return from function

Return from a routine is performed by the `RTRN` instruction. This restores the previous value of `P` and resumes execution from the return address. If the return is from a function, the `FNRN` instruction is planted just after the

result has been evaluated (this is always placed on the top of the stack). The `FNRN` instruction is identical to `RTRN` after it has stored the result in the `A` register ready for the `FNAP` instruction to store it at the required location in the previous stack frame.

2.5.4 Control Instructions

Control instructions are to be found in most virtual machines. Their function is centred around the transfer of control from one point to another in the code. Included in this set are instructions to create labels in code. The OCODE control instructions are shown in Figure 2.4.

Table 2.4. *OCODE control instructions.*

Opcode	Meaning
`LAB` Ln	declare label
`JUMP` Ln	unconditionally jump to label
`JT` Ln	jump if top of stack is true
`JF` Ln	jump if top of stack is false
`GOTO` E	computed goto (see below)
`RES` Ln	return
`RSTACK` k	return
`SWITCHON` n Ld K_1 $L_1 \ldots L_n$	jump table for a SWITCHON.
`FINISH`	terminate execution

The `JUMP Ln` instruction transfers control unconditionally to the label `L`. The instructions `JT` and `JF` transfer control to their labels if the top of the stack (implemented as `P!(S-1)`) is true or false, respectively. Instructions like this are often found in the instruction sets of virtual machines. The conditional jumps are used, *inter alia*, in the implementation of selection and iteration commands.

Although they are particular to OCODE, the other instructions also represent typical operations in a virtual machine. The `LAB` instruction (really a pseudo-operation) declares its operand as a label (thus associating the address at which it occurs with the label).

The `GOTO` instruction is used to generate code for SWITCHON commands. It takes the form `GOTO` E, where E is an expression. In the generated code, the code for E is compiled and immediately followed by the `GOTO` instruction. At runtime, the expression is evaluated, leaving an address on the top of the stack. The `GOTO` instruction then transfers control to that address.

The `RES` and `RSTACK` instructions are used to compile RESULTIS commands. If the argument to a RESULTIS is immediately returned as the result of a function, the `FNRN` instruction is selected. In all other contexts, RESULTIS e compiles to the code for e followed by the `RES` Ln instruction. The execution of this instruction places the result in the `A` register and then jumps to

the label `L`n. The label addresses an `RSTACK` k instruction, which takes the result and stores it at location `P!`k and sets `S` to $k+1$.

The OCODE `SWITCHON` instruction performs a jump based on the value on the top of the stack. It is used to implement switches (SWITCHON commands, otherwise known as case statements). It has the form shown in Table 2.4, where n is the number of cases to which to switch and `L`d is the label of the default case. The K_i are the case constants and the L_i are the corresponding code labels.

Finally, the `FINISH` instruction implements the BCPL FINISH command. It compiles to `stop(0)` in code and causes execution to terminate.

2.5.5 Directives

It is intended that BCPL programs be compiled to OCODE (or native code) and then executed in their entirety. The BCPL system is not intended to be incremental or interactive. It is necessary, therefore, for the compiler to provide information to the runtime system that relates to the image file that it is to execute. This is the role of the directives.

The BCPL OCODE machine manages a globals area, a stack and a code segment. The runtime system must be told how much space to allocate to each. It must also be told where globals are to be located and where literal pools start and end, so that modules can be linked. The system also needs to know which symbols are exported from a module and where modules start and end.

The BCPL global vector is a case in point. There is no *a priori* limit on the size of the global vector. In addition, two modules can assign different values to a particular cell in the global vector (with all the ordering problems that are so familiar).

The OCODE generator also needs to be handed information in the form of directives. The directives in the version of BCPL that is current at the time of writing (Summer, 2004) are as shown in Table 2.5. The directives are used in different parts of the system, so are briefly explained in the following few paragraphs.

Table 2.5. *OCODE directives.*

Directive
`STACK` s
`STORE`
`ITEMN` n
`DATALAB L`n
`SECTION`
`NEEDS`
`GLOBAL` n $K_1L_1 \ldots K_nL_n$

The `STACK` directive informs the code generator of the current size of the stack. This is required because the size of the current stack frame can be affected by some control structures, for example those that leave a block in which local variables have been declared.

The `STORE` directive informs the code generator that the point separating the declarations and code in a block has been reached. Any values left on the stack are to be treated as variable initialisations and should be stored in the appropriate places.

Static variables and tables are allocated in the program code area using the `ITEMN` directive. The parameter to this directive is the initial value of the cell that is reserved by this directive. For a table, the elements are allocated by consecutive `ITEMN` directives. The `DATALAB` directive is used to associate a label with a data area reserved by one or more `ITEMN` directives.

The `SECTION` and `NEEDS` directives are direct translations of the SECTION and NEEDS source directives. The latter are used to indicate the start of a BCPL module and the modules upon which the current one depends.

An OCODE module is terminated with the `GLOBAL` directive. The arguments denote the number of items in the global initialisation list and each of the K_i are offsets into the global vector and L_n is the label of the corresponding offset (i.e., K_iL_i denotes an offset and the label to be associated with that offset).

Directives are an important class of virtual machine instruction, although little more will be said about them. One reason for this is that, once one becomes aware of their need, there is little else to be said. A second reason is that, although every system is different, there are things that are common to all—in this case, the general nature of directives. It is considered that the directives required by any virtual machine will become clear during its specification.

2.6 The Intcode/Cintcode Machine

The Intcode/Cintcode machine is used to bootstrap an OCODE machine on a new processor; it can also serve as a target for the BCPL compiler's code generator. The code is designed to be as compact as possible. The Cintcode machine was originally designed as a byte-stream interpretive code to run on small 16-bit machines such as the Z80 and 6502 running under CP/M. More recently, it has been extended to run on 32-bit machines, most notably machines running Linux.

The best descriptions of the Intcode and Cintcode machines are [45] and [44], respectively. Compared with OCODE, (Ci/I)ntcode is an extremely compact representation but is somewhat more complex. The complexity arises because of the desire to make the instruction set as compact as possible; this is reflected in the organisation which is based on bit fields. The organisation of the machine is, on the other hand, easily described. The following description

is of the original Intcode machine and follows that in [45] (the account in [44] is far more detailed but is essentially the same in intent).

The Intcode machine is composed of the following components. A memory consisting of equal-sized locations that can be addressed by consecutive integers (a vector of words, for example). It has a number of central registers:

A,**B**: the *accumulator* and *auxiliary accumulator*;
C: the *control register*. This is the instruction pointer; it points to the next instruction to be executed;
D: the *address register*, used to store the effective address of an instruction;
P: a pointer that is used to address the *current stack frame*;
G: a pointer used to access the global vector.

Note that the Intcode machine has a framed stack and a global vector (both necessary to implement OCODE).

Instructions come in two lengths: single and double length. The compiler determines when a double-length instruction should be used.

The operations provided by the Intcode machine are shown in Table 2.6 (the idea is taken from [45], p. 134; the specification has been re-written using mostly C conventions). As in the OCODE instructions, each operation is specified by a code fragment.

Table 2.6. *The Intcode machine functions.*

Operation	Mnemonic	Specification
Load	**L**	`B := A:  A := D`
Store	**S**	`*D := A`
Add	**A**	`A := A + D`
Jump	**J**	`C := D`
Jump if true	**T**	`IF A THEN C := D`
Jump if false	**F**	`IF NOT A THEN C := D`

Call routine	**K**	`D := P + D`
		`*D := P; *(D+1) := C`
		`P := D; C := A`

Execute operation	**X**	Various operations, mostly arithmetic of logical operations operating on `A` and `B`.

Each Intcode instruction is composed of six fields. They are as follows:

- Function Part: This is a three-bit field. It specifies one of the eight possible machine operations described in Table 2.6.
- Address Field: This field holds a positive integer. It represents the initial value of the `D` register.
- D bit: This is a single bit. When set, it specifies that the initial value of the `D` register is to be taken from the following word.

- P bit: This is single bit. It specifies whether the `P` register is to be added to the `D` register at the second stage of an address calculation.
- G bit: This is another single bit field. It specifies whether the `G` register is to be added to the `D` register at the end of the third stage of address calculation.
- I bit: This is the *indirection* bit. If it is set, it specifies that the `D` register is to be relaced by the contents of the location addressed by the `D` register at the last stage of address calculation.

The effective address is evaluated in the same way for every instruction and is not dependent upon the way in which the machine function is specified.

Intcode is intended to be a compact representation of a program. It is also intended to be easy to implement, thus promoting BCPL's portability (the BCPL assembler and interpreter for Intcode occupies just under eight and a half pages of BCPL code in [45]).

The Intcode machine also uses indirection (as evidenced by the three-stage address calculation involving addresses in registers), thus making code compact.

This has, of necessity, been only a taster for the Intcode and Cintcode machines. The interested reader is recommended to consult [44] and [45] for more information. The full BCPL distribution contains the source code of the OCODE and Cintcode machines; time spent reading them will be rewarding.

http://www.springer.com/978-1-85233-969-2

Virtual Machines
Craig, I.D.
2006, XV, 269 p. 43 illus., Hardcover
ISBN: 978-1-85233-969-2